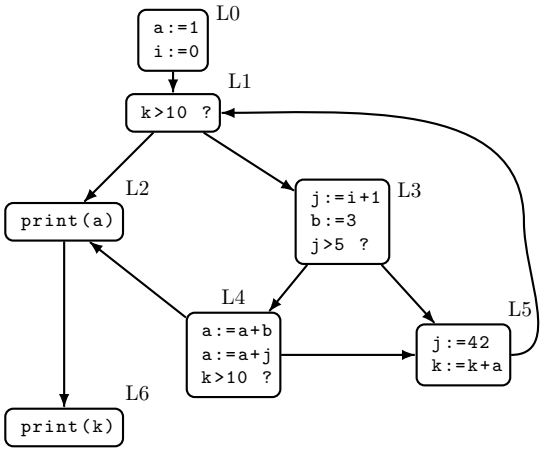
**Master Info - 2016-2017**
**MIF08 Compilation**

# Contrôle continu 1 - Durée 20 min Éléments de correction

## 1 Dataflow

On donne un graphe de flot, et on rappelle qu'*une variable est vivante en sortie d'un bloc si il existe un chemin partant de ce bloc qui mène à une utilisation de cette variable sans redéfinition de cette variable.*




| bloc | variables vivantes en sortie du bloc |
|---|---|
| L0 | |
| L1 | |
| L2 | |
| L3 | |
| L4 | |
| L5 | |
| L6 | $\emptyset$ |

**Question 1** (2 points)
Sans calcul, remplir le tableau avec les variables vivantes en sortie de chaque bloc.

**Solution:**
Voici le tableau rempli :

| bloc | Out(bloc) |
|---|---|
| L0 | a,i, k |
| L1 | a,i,k |
| L2 | k |
| L3 | i,j,a,k,b |
| L4 | a,i,k |
| L5 | a,i, k |
| L6 | $\emptyset$ |

Il y avait plusieurs petites difficultés :

- $k$ est utilisé dans le programme, ce qui sous-entend qu'il a été défini précédemment.
- $b$ est bien vivant en sortie du bloc L3 (sa valeur est utilisée juste après)
- lorsque j'ai une instruction d'impression, la variable utilisée doit être vivante en entrée du bloc...

J'ai compté des points pour des solutions partielles.

**Question 2** (1 point)
Peut-on déplacer l'affectation `b:=3` de L3 à L0 ? Pourquoi ? Quel intérêt cela peut-il avoir ?

**Solution:**
On peut déplacer cette affectation avant la boucle, car $b$ n'est définie qu'une seule fois (et L0 domine L3, c'est à dire que tout chemin passant par L3 est forcément passé par L0). Cela permet de ne pas recalculer $b$ à chaque tour de boucle, donc accélère le programme.
j'ai mis des points pour "diminue la pression registre dans la boucle"

**Question 3** (2 points)
Enlever le code mort, en justifiant.

**Solution:**
L'instruction j :=42 est morte dans le bloc L5, car j n'est pas vivante en sortie de L5.
Je n'attendais que cette réponse qui provient d'une utilisation directe du tableau. J'ai compté des points aux remarques sur le fait que l'on peut montrer que j est une constante, donc montrer que L4 n'est jamais accessible. Néanmoins, ce n'est pas une conséquence de l'analyse des variables vivantes, mais d'une analyse qui propagerait les constantes.
Attention, `a:=a+b` ne peut pas être supprimée car la valeur de $a$ est modifiée ...

## 2 Génération de code

On rappelle en annexe les règles de génération de code pour les affectations, le test... Le visiteur de génération de code développé durant le TP4 a été lancé sur l'entrée suivante :

```
x=2;
if (x<3) y=7; else  y=8;
z=y+1;
```

**Question 4** (5 points)
*En vous aidant des règles en annexe*, compléter le code généré suivant, en considérant l'allocation de temporaires suivante $x \mapsto tmp_1$, $y \mapsto tmp_6$, $z \mapsto tmp_9$ :

---

**Solution:**
Voici le code généré complet :

*;;Automatically generated code*
*;;( stat (assignment x = (expr (atom 2)) ;))*
**AND** temp_1 , temp_1 , 0
**ADD** temp_1 , temp_1 , 2
*;;( stat ( if_stat if ( condition_block (expr (atom ( (expr (expr (atom x)) < (expr ( atom 3))) ))) ( stat_block { (block (stat (assignment y = (expr (atom 7)) ;))) })) else ( stat_block { (block (stat (assignment y = (expr (atom 8)) ;))) }))*
**AND** temp_2 , temp_2 , 0
**ADD** temp_2 , temp_2 , 3
**NOT** temp_4 , temp_2
**ADD** temp_4 , temp_4 , 1
**ADD** temp_3 , temp_1 , temp_4
**BRzp** l_cond_neg_2

**AND** temp_5 , temp_5 , 0
**ADD** temp_5 , temp_5 , 1
**BR** l_cond_end_2
l_cond_neg_2:
**AND** temp_5 , temp_5 , 0
l_cond_end_2:
**BRz** l_if_false_3
*;;( stat (assignment y = (expr (atom 7)) ;))*
**AND** temp_6 , temp_6 , 0
**ADD** temp_6 , temp_6 , 7
**BR** l_if_end_1
l_if_false_3 :
*;;( stat (assignment y = (expr (atom 8)) ;))*
**AND** temp_7 , temp_7 , 0
**ADD** temp_7 , temp_7 , 8
**ADD** temp_6 , temp_7 , 0
l_if_end_1 :
*;;( stat (assignment z = (expr (expr (atom y)) + (expr (atom 1))) ;))*
**AND** temp_8 , temp_8 , 0
**ADD** temp_8 , temp_8 , 1
**ADD** temp_9 , temp_6 , temp_8

Faites attention à générer le code en n'oubliant pas les temporaires générés, même s'ils sont inutiles ; en particulier, lors dans la seconde branche du test, même si y est affecté à $temp_6$ on ne peut pas directement mettre la constante dans $temp_6$. De même, lors de la génération de test pour l'addition il faut bien créer un nouveau temporaire pour la constante 1.

Il y a très peu de code correct pour les tests.

| | |
|---|---|
| e ::= c | `dr <-newTemp()`<br>`code.add(InstructionAND(dr, dr, 0))`<br>`code.add(InstructionADD(dr, dr, c))`<br>`return dr` |
| e ::= x | `regval=getTemp(x) #get the place associated to x.`<br>`return regval` |
| e ::= $e_1$-$e_2$ | `t1 <- GenCodeExpr(e_1)`<br>`t2 <- GenCodeExpr(e_2)`<br>`dr <- newTemp()`<br>`code.add(InstructionNOT(dr, t2))`<br>`code.add(InstructionADD(dr, dr, 1))`<br>`code.add(InstructionADD(dr, dr, t1))`<br>`return dr` |
| e ::= $e_1 < e_2$ | `dr <-newTemp()`<br>`t1 <- GenCodeExpr (e1-e2)  #last write in register`<br>`(lfalse,lend) <- newLabels()`<br>`code.add(InstructionBRzp(lfalse))  #if =0 or >0 jump!`<br>`code.add(InstructionAND(dr, dr, 0))`<br>`code.add(InstructionADD(dr, dr, 1))  #dr <- true`<br>`code.add(InstructionBR(lend))`<br>`code.addLabel(lfalse)`<br>`code.add(InstructionAND(dr, dr, 0))  #dr <- false`<br>`code.addLabel(lend)`<br>`return dr` |
| x := e | `dr <- GenCodeExpr(e)`<br>`#a code to compute e has been generated`<br>`if x has a location loc:`<br>`  code.add(instructionADD(loc,dr,0))`<br>`else:`<br>`  storeLocation(x,dr)` |
| if $b$ then $S1$ else $S2$ | `dr <-GenCodeExpr(b)  #dr is the last written register`<br>`lfalse,lendif=newLabels()`<br>`code.add(InstructionBRz(lfalse) #if 0 jump to execute S2`<br>`GenCodeSmt(S1)  #else (execute S1`<br>`code.add(InstructionBR(lendif)) #and jump to end)`<br>`code.addLabel(lfalse)`<br>`GenCodeSmt(S2)`<br>`code.addLabel(lendif)` |

FIGURE 1 – Code generation rules